KaSim reference manual v3

KappaLanguage.org

March 20, 2012

This document is work in progress...

Contents

Chapter 1

Introduction

1.1 Preamble

This manual describes the usage of KaSim, the latest implementation of Kappa, one member of the growing family of rule-based languages. Rule-based modelling has attracted recent attention in developing biological models that are concise, comprehensible, and easily extensible. Although this manual contains a self-contained description of Kappa, it is *not* intended as a tutorial on rule-based modeling.[1]

1.2 The KaSim engine

KaSim is an open source stochastic simulator of rule-based models [3, 2, 4] written in the κ-calculus. Basically KaSim takes one or several kappa files as input and generates stochastic trajectories of various observables. KaSim implements Danos and Krivine's simulation algorithm [1] which adapts Gillespie's algorithm [5, 6] to rule-based models.

[1] For an idea of how Kappa is used in a modeling context, the reader might find the following short note useful Agile modelling of cellular signalling (SOS'08). A longer article, expounding on the causal aspects of modeling, is also available: Rule-based modelling of cellular signalling (CONCUR'07), See also this tutorial: Modelling epigenetic information maintenance: a Kappa tutorial (CAV'09).

A *simulation event* corresponds to the application of a rewriting rule, contained in the kappa files, to the current graph (also called a *mixture*). The rule is selected according to its *activity*, ie the number of instances it has in the current mixture, multiplied by its kinetic rate, and applied to one of its possible instances in the graph. It results in a new graph together with an updated activity for all rules (see Fig. 1.1). Importantly, the cost of a simulation event is independent of the size of the graph it is applied to [1].

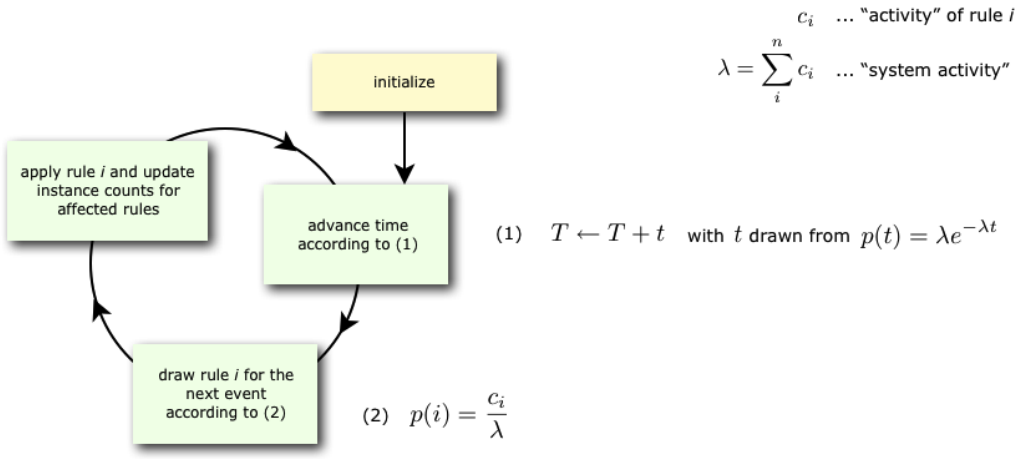

Figure 1.1: The event loop

Note that KaSim is not equipped with a curve visualization tool. However, data outputs are given in a text format that is readily usable by any standard plotting software such as gnuplot.

1.3 Support

- Kappa language tutorials and downloads: `http://kappalanguage.org`
- Bug reports should be posted on github: `https://github.com/jkrivine/KaSim/issues`
- Questions and answers on the kappa-user mailing list: `http://groups.google.com/group/kappa-users`
- Want to contribute to the project? `jkrivine` at `pps` dot `jussieu` dot `fr`

Chapter 2

Installation

2.1 Using precompiled binaries

The easiest way to use KaSim is to use pre-compiled versions available at `https://github.com/jkrivine/KaSim/downloads`. Download the version that corresponds to your operating system (Windows, Linux or Mac OSX) and rename the downloaded file into KaSim. Note that on Mac OSX or linux it might be necessary to give executable permissions to KaSim. This can be done using the shell command: `chmod u+x KaSim`.

To test whether your program does work, simply type `./KaSim --version` on a terminal, from the directory that contains the binaries. If the version is displayed it means that the binaries are indeed compatible with your OS. Otherwise, check that you gave the executable permissions to KaSim (see above). Otherwise you may need to compile KaSim from the sources (see next Section).

2.2 Obtaining the sources

To obtain KaSim you can either use pre-compiled binaries (see previous section) or compile the sources for your architecture. To do so, download the source code from `https://github.com/jkrivine/KaSim` and make sure you have a recent ocaml compiler installed. From a terminal window type `ocamlopt.opt -v`. If nothing appears then you need to install Ocaml Native compiler that can be downloaded from `http://caml.inria.fr/download.en.html`.

2.3 Compilation

Once Ocaml is safely installed, untar KaSim archive and compile following these few steps:

```
$ tar xzvf kasim.tar.gz -d Kappa
```

```
$ cd Kappa
```

```
$ make
```

At the end of these steps you should see, in the `Kappa` directory, an executable file named KaSim. In order to check the compilation went fine, simply type `.\KaSim --version`. If the ocaml native compiler `ocamlopt.opt` is not in the path of your system, you may set the variable `OCAMLBINPATH` to point to the location of the compiler by editing the corresponding line in the Makefile.

Chapter 3

The command line

3.1 General usage

From a terminal window, KaSim can be invoked by typing

```
$ KaSim -i file_1 ... -i file_n [option]
```

where `file_i` are the input kappa files containing the rules, initial conditions and observables, see Chapter 4 below. Tables 3.1 and 3.2 summarize all the options that can be given to the simulator. Basically, one specifies an upper bound either in bio time (arbitrary time unit), or in number of events. Note that bio-time is computed using Gillespie's formula for time advance (see Fig. 1.1) and should not be confused with CPU-time (it's not even proportional). In doubt, we recommend using a bound in number of events since the cost of one event application is bounded (in CPU time) by a constant, so the simulation time of n events is roughly k times faster than a simulation of $k * n$ events.

3.2 Main options

Table 3.1 summarizes the main options that are accessible through the command line. Options that expects an argument are preceded by a single dash, options that do not need any argument start with a double dash. Note that the option `-p` specifies the number of points that one wishes to have in the final plot. The interval at which these points will be taken is then computed using the simulation limit defined by the user using option `-t` or `-e`. For instance requiring a simulation of 100 points during 10 time units will result in a simulation where observables are recorded every 0.1 time units, while requiring 100 points for 1000 events will result in observable being written every 10 events (be careful that the event density, ie the number of events per time unit, might vary during a simulation, and thus the two sampling methods can lead to very different repartitions of the moments at which observables are recorded even if they sepecify the same number of observations).

The results of the options `-im` and `-flux` is explained in Chapter 6.

Table 3.1: Command line: main options

Argument	Description
`-e` e_{max}	Terminates simulation after $e_{max} \geq 0$ events
`-t` t_{max}	Terminates simulation after $t_{max} \geq 0.0$ time units
`-p` n	Produces a data file (default: `data.out`) with $n \geq 0$ data points
`-o` *file*	Set the name of data file to *file*
`-d` *dir*	Output any produced file to the directory *dir*

3.3 Advanced options

Table 3.2 summarizes the advanced options that are accessible through the command line.

Table 3.2: Command line: advanced options

Argument	Description
`-seed` n	Seeds the pseudo-random number generator $n > 0$
`--implicit-signature`	Automatically deduce agent signatures (see Chapter 6)
`-im` *file*	Produces a dot format file of the decorated influence map
`-flux` *file*	Produces a dot format file of the activation/inhibition flux measured during the simulation
`-make-sim` *sim_file*	makes a simulation package out of the input kappa files
`-load-sim` *sim_file*	use simulation package *sim_file* as input
`--gluttony`	simulation mode that is memory intensive but that speeds up simulation time

Example

The command

```
$ KaSim -i model.ka -e 1000000 -p 1000 -o model.out
```

will generate a file `model.out` containing the trajectories of the observables defined in the kappa file `model.ka`. The file `model.out` will contain 1000 data points (ie in this case, a measure will be taken every 1000 events). The command

```
$ KaSim -i init.ka -i rules.ka -i obs.ka -i mod.ka -t 1.5 -p 1000
```

will generate a file `data.out` (default name) containing 1000 data points of a simulation of 1.5 seconds (arbitrary time units) of the model. Note that the input kappa file is split in 4 files containing, for instance, the initial conditions, `init.ka`, the rule set, `rules.ka`, the observables, `obs.ka`, and the perturbations, `pert.ka` (see Chapter 4 for details). The order in which the files are given does not matter.

The activation and flux maps

The command line options `-im` and `-flux` allows modelers to generate additional information about the statics and the dynamics of the model that is being considered (see Chapter 6).

Chapter 4

The kappa file

4.1 General remarks

The *Kappa File* (KF) is the formal representation of your model. We use KF to denote the union of the files that are given as input to KaSim (argument `-i`). Each line of the KF is interpreted by KaSim as a *declaration*. If the line is ended by the escape character '\' the continuation of the declaration is parsed onto the next line. Declarations can be of 5 types: *signatures* (Sec. 4.2), *rules* (Sec. 4.3), *variables* (Sec. 4.4), *initial conditions* (Sec. 4.5), *perturbations* (Sec. 6.1) and *parameter configurations* (Sec. 6.7). The KF's structure is quite flexible and can be divided in any number of sub-files in which the order of declarations does not matter (to the exception of variable declarations, see Section 4.4 for details). Comments can be used by inserting the marker `#` that tells KaSim to ignore the rest of the line.

4.2 Agent signature

Agent signatures constitute a form of typing information about the agents that are used in the model. It contains information about the name and number of interaction sites the agent has, and about their possible internal states. A signature is declared in the KF by the following line:

`%agent:` *signature_ expression*

according to the grammar given in Table 4.1. For instance the line:

```
%agent: A(x,y~u~p,z~0~1~2) # Signature of agent A
```

will declare an agent `A` with 3 *(interaction) sites* `x`,`y` and `z` with the site `y` possessing two *internal states* `u` and `p` (for instance for the unphosphorylated and phosphorylated forms of `y`) and the site `z` having possibly 3 states respectively 0, 1 and 2. Note that internal states values are treated as untyped symbols by KaSim, so choosing a character or an integer as internal state is purely matter of convention.

signature_expression	::=	`Id(`*sig*`)`
sig	::=	`Id` *internal_state_list* `,` *sig* $\mid \varepsilon$
internal_state_list	::=	`~Id` *internal_state_list* $\mid \varepsilon$

Table 4.1: Agent signature expression: terminal symbol are denoted in (blue) typed font. Symbol `Id` can be any string generated by regular expression [*a-z A-Z* 0-9][*a-z A-Z* 0-9 _ −]*. Terminal symbol ε stands for the empty symbol.

4.3 Rules

Once agents are declared, one may add to the KF the rules that describe their dynamics through time. Roughly a Kappa rule looks like

`'my rule'` *kappa_expression* $\rightarrow$ *kappa_expression* `@` *rate_expression*

where `'my rule'` can be any name that will refer to the subsequent rule that can be decomposed into a *left hand side* (LHS) and a *right hand side* (RHS) kappa expressions together with a *kinetic rate expression* . Kappa and rate expressions are generated by the grammar given in Table 4.2.

kappa_expression	::=	*agent_expression* `,` *kappa_expression* $\mid \varepsilon$
agent_expression	::=	`Id(`*interface*`)`
interface	::=	ε $\mid$ `Id` *internal_state link_state*
internal_state	::=	ε $\mid$ `~Id`
link_state	::=	ε $\mid$ `!n` $\mid$ `!_` $\mid$ `?`
rate_expression	::=	γ $\mid$ γ`(`k`)`

Table 4.2: Kappa expressions: In addition to the conventions of Table 4.1, symbol `n` denotes any positive integer and γ, k are any positive real number.

4.3.1 A simple rule

With the signature of `A` defined in the previous section, the line

`'A dimerization' A(x),A(y~p)` $\rightarrow$ `A(x!1),A(y~p!1) @` γ

denotes a dimerization rule between two instances of agent `A` provided the second is phosphorylated (say that is here the meaning of `p`) on site `y`. Note that the bond between both `A`s is denoted by the identifier `!1` which uses an arbitrary integer (`!0` would denote the same bond. In Kappa, a bond may connect exactly 2 sites so any occurrence of a bond identifier `!`n has to be paired with exactly one other sibling in the expression. Note also the fact that site `z` of `A` is not mentioned in the expression which means that it has no influence on the

triggering of this rule. This is the *don't care don't write convention* (DCDW) that plays a key role in resisting combinatorial explosion when writing models.

4.3.2 Adding and deleting agents

Sticking with `A`'s signature, the rule

`'budding A' A(z)` $\rightarrow$ `A(z!1),A(x!1) @` γ

indicates that an agent `A` free on site `z`, no matter what its internal state is, may beget a new copy of `A` bound to it via site `x`. Note that in the RHS, agent `A`'s interface is not completely described. Following the DCDW convention, KaSim will then assume that the sites that are not mentioned are created in the *default state*, ie they appear free of any bond and their internal state (if any) is the first of the list shown in the signature (here state `u` for `y` and `0` for `z`).

Importantly, KaSim respects the *longest prefix convention* to determine which agent in the RHS stems from an agent in the LHS. In a word, from a rule of the form $a_1, \ldots, a_n \rightarrow b_1, \ldots, b_k$, with a_is and b_js being agents, one computes the biggest indices $i \leq n$ such that the agents $a_1, \ldots, a_i$ are pairwise consistent with $b_1, \ldots, b_i$, ie the a_js and b_js have the same name and the same number of sites. In which case we say that the for all $j \leq i$, a_j is *preserved* by the transition and for all $j > i$, a_j is *deleted* by the transition and b_j is *created* by the transition. This convention allows us to write a deletion rule as:

`'deleting A' A(x!1),A(z!1)` $\rightarrow$ `A(x) @` γ

which will remove the `A` agent in the mixture that will match the second occurrence of `A` in this rule.

4.3.3 Side effects

It may happen that the application of a rule has some *side effects* on agents that are not mentioned explicitly in the rule. Consider for instance the previous rule:

`'deleting A' A(x!1),A(z!1)` $\rightarrow$ `A(x) @` γ

The `A` in the graph that is matched to the second occurrence of `A` in the LHS will be deleted by the rule. As a consequence all its sites will disappear together with the bonds that were pointing to them. For instance, when applied to the graph

$G =$`A(x!1,y~p,z~2),A(x!2,y~u,z~0!1),C(t!2)`

the above rule will result in a new graph $G' =$ `A(x!1,y~p,z~2),C(t)` where the site `t` of `C` is now free as side effect.

Wildcard symbols for link state `?` (for bound or not), `!_` (for bound to someone), may also induce side effects when they are not preserved in the RHS of a rule, as in

`'Disconnect A' A(x!_)` $\rightarrow$ `A(x) @` γ

or

`'Force bind A' A(x?)` $\rightarrow$ `A(x!1),C(t!1) @` γ

Both these rule will cause KaSim to raise a warning at rule compilation time.

4.3.4 Rates

As said earlier, κ rules are equipped with one or two *kinetic rate(s)*. A rate is a real number called the *individual-based or stochastic rate constant*, it is the rate at which the corresponding rule is applied per instance of the rule. Its dimension is the inverse of a time $[T^{-1}]$.

The stochastic rate is related to the *concentration-based rate constant* k of the rule of interest by the following relation:

$$k = \gamma(\mathcal{A}\, V)^{(a-1)} \tag{4.1}$$

where V is the volume where the model is considered, $\mathcal{A} = 6.022 \cdot 10^{23}$ is Avogadro's number, $a \geq 0$ is the arity of the rule (ie 2 for a bimolecular rule).

In a modeling context, the constant k is typically expressed using *molars* $M := moles\, l^{-1}$ (or variants thereof such as μM, nM), and seconds or minutes. If we choose molars and seconds, k's unit is $M^{1-a} s^{-1}$, as follows from the relation above.

Concentration-based rates are usually favored for measurements and/or deterministic models, so it is useful to know how to convert them into individual-based ones used by KaSim. Here are typical volumes used in modeling:

- Mammalian cell: $V = 2.25\ 10^{-12} l$ ($1l = 10^{-3} m^3$), and $\mathcal{A}V = 1.35\ 10^{12}$.

  A concentration of $1M$ in a mammalian cell volume corresponds to $1.35\ 10^{12}$ molecules; $1nM \approx 1350$ molecules per cell.

- Yeast cell (haploid): $V = 4\ 10^{-14} l$, and $\mathcal{A}V = 2.4\ 10^{10}$.

  A concentration of $1M$ in a yeast cell volume corresponds to $2.4\ 10^{10}$ molecules; $1nM \approx 24$ molecules per cell. The volume is doubled in a diploid cell.

- E. Coli cell: $V = 10^{-15} l$, and $\mathcal{A}V = 10^8$.

  A concentration of $1M$ in a yeast cell volume corresponds to 10^8 molecules; $10nM \approx 1$ molecule per cell.

The table below lists typical ranges for deterministic rate constants and their stochastic counterparts assuming a mammalian cell volume.

process	k	γ
general binding	$10^7 - 10^9$	$10^{-5} - 10^{-3}$
general unbinding	$10^{-3} - 10^{-1}$	$10^{-3} - 10^{-1}$
dephosphorylation	1	1
phosphorylation	0.1	0.1
receptor dimerization	$2\ 10^6$	$1.6\ 10^{-6}$
receptor dissociation	$1.6\ 10^{-1}$	$1.6\ 10^{-1}$

4.3.5 Ambiguous molecularity

It is considered malpractice to use a κ-rule of the form `A(x),B(y)`$\rightarrow \ldots$ `@` γ in a model where this rule could be applied in a context where `A` and `B` are sometimes already connected and sometimes disconnected. Indeed, this would lead to an inconsistency in the definition of the kinetic rate γ which should have a volume dependency in the former case and no volume dependency in the latter (see Section 4.3.4).

This sort of ambiguity should be resolved, if possible, by refining the ambiguous rule into cases that are either exclusively unary or binary. Each refinement having a kinetic rate that is consistent with its molecularity. Note that in practice, for models with a large number of agents, it is sufficient to assume that the rule `A(x),B(y)`$\rightarrow \ldots$ `@` γ will have only binary instances. In this case it suffices to consider the approximate model:

`'assumed binary AB' A(x),B(y)`$\rightarrow \ldots$ `@` γ_2

`'unary AB' A(x,c!1),C(a!1,b!2),B(y,c!2)`$\rightarrow \ldots$ `@` k_1

There are however systems where even enumerating unary cases becomes impossible. As an alternative, one should use the kappa notation for ambiguous rules:

`'my rule'` *kappa_ expression* $\rightarrow$ *kappa_ expression* `@` $\gamma_2(k_1)$

which will tell KaSim to apply the above rule with a rate γ_2 for binary instances and a rate k_1 for unary instances. The obtained model will behave exactly as a model in which the ambiguous rule has been replaced by unambiguous refinements. However the usage of such rule might *slowdown simulation in a significant manner* depending on various parameters (such as the presence of large polymers in the model). We give below an example of a model utilizing binary/unary rates for rules[1].

```
1. %agent: A(b,c)
2. %agent: B(a,c)
3. %agent: C(b,a)
4. ##
```

[1]This model is available in the source repository `model/poly.ka`.

```
 5. %var: 'V' 1
 6. %var: 'k1' [inf]
 7. %var: 'k2' 1.0E-4/'V'
 8. %var: 'k_off' 0.1
 9. ##
10. 'a.b' A(b),B(a) -> A(b!1),B(a!1) @ 'k2'('k1')
11. 'a.c' A(c),C(a) -> A(c!1),C(a!1) @ 'k2'('k1')
12. 'b.c' B(c),C(b) -> B(c!1),C(b!1) @ 'k2'('k1')
13. ##
14. 'a..b' A(b!a.B) -> A(b) @ 'k_off'
15. 'a..c' A(c!a.C) -> A(c) @ 'k_off'
16. 'b..c' B(c!b.C) -> B(c) @ 'k_off'
17. ##
18. %var: 'n' 1000
19. ##
20. %init: 'n' A,B,C
21. %mod: [E] > 10000 do $STOP
```

Notice at lines 10-12 the use of binary/unary notation for rules. As a result binding between between freely floating agents will occur at rate `'k2'` while binding between agents that are part of the same complex will occur at rate `'k1'`. Line 21 contains a *perturbation* that requires KaSim to stop the simulation after 10,000 events and output the list of molecular species present in the final mixture (see Section 6.1) and that we give Figure 4.1.

4.4 Variables

In the KF it is also possible to declare *variables* by a line of the form

`%var: 'var_name'` (*variable_ expression* or *kappa_ expression*)

where `var_name` can be any string and *variable_ expression* is any algebraic expression on variable names (other than `var_name`) using predefined operators summarized in Table 4.3. For instance the declarations

```
%var: 'homodimer' A(x!1),A(x!1)

%var: 'aa' 'homodimer'/2
```

define 2 variables, the first one tracking the number of embeddings of `A(x!1),A(x!1)` in the graph over time, while the second divides this value by 2: the number of automorphisms in `A(x!1),A(x!1)`. The declaration of a variable acts as a binder in the rest of the KF. Hence variables that are used in the expression of another variable must be declared beforehand.

It is also possible to use algebraic expressions as kinetic rates as in

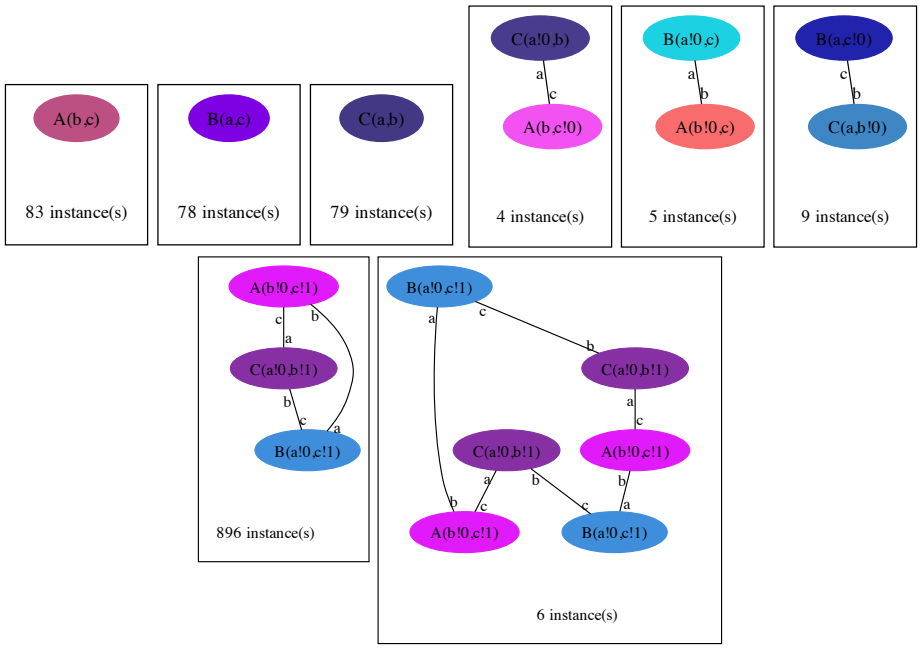

Figure 4.1: Final mixture obtained after 10,000 events of simulation of the `poly.ka` model. The infinite rate for cycle closure allows one to obtain a large number of triangles.

Symbol	Interpretation
`[E]`	the total number of simulation events since the beginning of the simulation
`[E+]`	the total number of productive events
`[E-]`	the total number of null events
`[emax]`	the max (productive) event limit as set by the option `-e`. Note that if unset `emax`$= \infty$
`[T]`	the bio-time of the simulation
`[Tsim]`	the cpu-time since the beginning of the simulation
`[tmax]`	the max (bio)-time limit as set by the option `-t`. Note that if unset `tmax`$= \infty$
`'v'`	the value of variable 'v' (declared using the `%var:` statement)
`[`f`]`	the intuitive mathematical function or constant associated to $f \in \{$`log,sin,cos,tan,sqrt,pi`$\}$
`[inf]`	symbol for ∞
`[mod]`	the *modulo* operator (infix notation)
`[exp]`	the exponentiation operation $x \mapsto e^x$
`[int]`	the floor function $x \in \mathbb{R} \mapsto \lfloor x \rfloor \in \mathbb{N}^+$
`+,-,*,/,^`	the corresponding mathematical operators (infix notation)

Table 4.3: Symbol usable in algebraic expressions.

```
%var: 'k_on' 1.0E-6 # per molecule per second

'ab' A(x),A(x) -> A(x!1),A(x!1) @ 'k_on'/2
```

KaSim may output values of variables in the data file (see option `-p` in Chapter 3) using plot declaration:

```
%plot: 'var_name'
```

One may use the shortcut:

`%obs: 'var_name'` *variable_ expression*

to declare a variable and at the same time require it to be outputted in the data file.

4.5 Initial conditions

The initial mixture to which rules in the KF will be applied are declared as

`%init:` *mult kappa_ expression*

where *mult* is either a positive integer or a variable name pointing to a constant expression (ie. not a κ expression). This will add to the initial state of the model *mult* copies of the graph described by the kappa expression. Again the DCDW convention allows us not to write the complete interface of added agents (the remaining sites will be completed according to the agent's signature). For instance:

```
%var: 'n' 1000

%init: 'n' (A,A(y~p))
```

will add 1000 instances of `A` in its default state `A(x,y~u,z~0)` and 1000 instances of `A` in state `A(x,y~p,z~0)`. Note that `A` is equivalent to writing `A()`. As any other declaration, `%init` can be used multiple times, and agents will add up to the initial state.

Chapter 5

A simple model

We describe below the content of a simple Kappa model and give examples of some typical run[1].

5.1 ABC.ka

```
1. #### Signatures
2. %agent: A(x,c) # Declaration of agent A
3. %agent: B(x) # Declaration of B
4. %agent: C(x1~u~p,x2~u~p) # Declaration of C with 2 modifiable sites
5. #### Rules
6. 'a.b' A(x),B(x) -> A(x!1),B(x!1) @ 'on_rate' #A binds B
7. 'a..b' A(x!1),B(x!1) -> A(x),B(x) @ 'off_rate' #AB dissociation
8. 'ab.c' A(x!_,c),C(x1~u) ->A(x!_,c!2),C(x1~u!2) @ 'on_rate' #AB binds C
9. 'mod x1' C(x1~u!1),A(c!1) ->C(x1~p),A(c) @ 'mod_rate' #ABC modifies x1
10. 'a.c' A(x,c),C(x1~p,x2~u) -> A(x,c!1),C(x1~p,x2~u!1) @ 'on_rate' #A binds C on x2
11. 'mod x2' A(x,c!1),C(x1~p,x2~u!1) -> A(x,c),C(x1~p,x2~p) @ 'mod_rate' #A modifies x2
12. #### Variables
13. %var: 'on_rate' 1.0E-4 # per molecule per second
14. %var: 'off_rate' 0.1 # per second
15. %var: 'mod_rate' 1 # per second
16. %obs: 'AB' A(x!x.B)
17. %obs: 'Cuu' C(x1~u,x2~u)
18. %obs: 'Cpu' C(x1~p,x2~u)
19. %obs: 'Cpp' C(x1~p,x2~p)
```

[1]The corresponding kappa file is included in the distribution of KaSim, in the directory `models/`

```
20. #### Initial conditions
21. %init: 1000 A,B
22. %init: 10000 C
```

Line 1-4 of this KF contains signature declarations. Agents of type `C` have 2 sites `x1` and `x2` whose internal state may be `u`(nphosphorylated) or `p`(hosphorylated). Recall that the default state of these sites is `u` (the first one). Line 8, rule `'ab.c'` binds an `A` connected to someone on site `x` (link type `!_`) to a `C`. Note that the only rule that binds an agent to `x` of `A` is `'a.b'` at line 6. Hence the use of `!_` is a commodity and the rule could be replaced by

`'alt_ab.c' A(x!1,c),B(x!1),C(x1~u)` $\rightarrow \ldots$

There are two main points to notice about this model: `A` can modify both sites of `C` once it is bound to them. However, only an `A` bound to a `B` can connect on `x1` and only a free `A` can connect on `x2`. Note also that `x2` is available for connection only when `x1` is already modified.

5.2 Some runs

We try first a coarse simulation of $100,000$ events (10 times the number of agents in the initial system).

```
$ KaSim -i ABC.ka -e 100000 -p 1000 -o abc.out
```

Plotting the content of the `abc.out` file one notices that nothing of significant interest happen to the observables after 250s. So we can now specify a meaningful time limit by running

```
$ KaSim -i ABC.ka -e 100000 -t 250 -p 1000 -o abc.out
```

which produces the data points whose rendering is given in Fig. 5.1. We will use this model as a running example for the next chapter, in order to illustrate various advanced concepts.

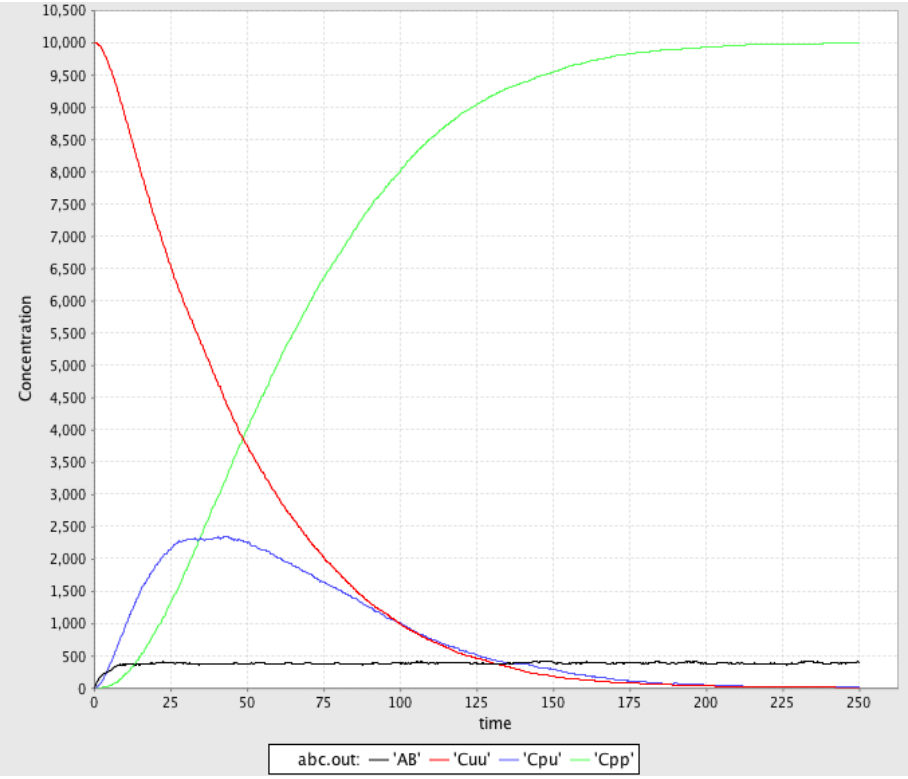

Figure 5.1: Simulation of the ABC model: population of unmodified `C`s (observable `Cuu` in red) drops rapidly and is replaced, in a first step by simply modified `C`s (observable `Cpu` in blue) which are in turn replaced by doubly modified `C`s (observable `Cpp` in red). Note that the population of `AB` complexes (observable `AB` in black) stabilizes slightly below 400 individuals after about 20s.

Chapter 6

Advanced concepts

6.1 Perturbation language

6.1.1 Syntax

It is possible to use variables of the model as precondition for triggering a *perturbation* of the simulation. For instance one may want to interrupt the simulation whenever a certain amount of observable has been reached, introduce a signal at some time interval etc. To do so, one declares a perturbation with a line:

`%mod:` *boolean_ expression* `do` *effect end_ expression*

where *boolean_ expression* and *effect* are defined by the grammar given in Table 6.1.

Precondition

Basically, the *precondition* determines when the perturbation will be triggered, for instance a user writes

```
%mod: ([T]>10) && (‘v1’/‘v2’) > 1 do ...
```

to indicate she wishes to trigger a perturbation whenever the simulation time has passed 10 time units and the ratio of variables `v1` over `v2` is above 1. Note that the perturbations are by default "one shot" interventions in the simulation. Once the precondition is satisfied, the perturbation is applied and discarded, unless a persistent perturbation is used using the `until` keyword (see below).

Effect

The *effect* of a perturbation can be divided in two kinds: the perturbations that modify the mixture, and those that simply give modelers an additional insight of what is going on in the simulation.

boolean_expression	::=	*algebraic_expression* `rel` *algebraic_expression* | (*boolean_expression* `||` *boolean_expression*) | (*boolean_expression* `&&` *boolean_expression*) | `[not]` *boolean_expression* | `[true]` | `[false]`
effect	::=	`$ADD` *algebraic_expression agent_expression* \| `$DEL` *algebraic_expression agent_expression* | `$SNAPSHOT` \| `$STOP` | `'rule_name' :=` *algebraic_expression* | `$TRACK` *observable*
observable	::=	`'rule_name'` | `'var_name'`
end_expression	::=	ε | `until` *boolean_expression*

Table 6.1: Perturbation expressions. The operator `rel` can be any usual binary relation in $\{<, =, >\}$ and algebraic expressions are built using table 4.3.

In the first kind one may add (`$ADD` instruction), delete (`$DEL` instruction) agents from the mixture or modify the kinetic rate of a particular rule. In the second category one may record the current state of the mixture (`$SNAPSHOT` instruction), interrupt the simulation in order to save the current state (`$STOP` instruction) or decide to trigger some causality analysis on some particular observable of interest (`$TRACK` instruction).

Persistent perturbation

The *end_expression* serves to specify whether the perturbation should be applied only once ("one shot" perturbation) or at every future events until a condition is reached (persistent perturbation). For instance:

```
%mod: ([T] [mod] 100)=0 do $DEL [inf] C until [T]>1000
```

will delete every `C` from the mixture every 100 t.u until the simulation clock has passed 1000 t.u. If no `until` keyword is positioned, the perturbation will be discarded after its first application.

6.1.2 Modifying the mixture during a simulation

Continuing with the ABC model, the perturbation effect:

```
$ADD n C(x1~p)
```

will add $n \geq 0$ instances of `C` with `x1` already in state `p` (and the rest of its interface in the default state as specified line 4 of ABC.ka). Also the perturbation effect:

```
$DEL [inf] B(x!_)
```

will remove *all* `B`s connected to someone from the mixture. Now, the effect:

```
'rule_name' := [inf]
```

will set the kinetic rate of a declared rule named `'rule_name'` to ∞.

Example

There are various ways one can use perturbations to study more deeply a given kappa model. A basic illustration is the use of a simple perturbation to let a system equilibrate before starting a real simulation. For instance, as can be seen from the curve given in Fig. 5.1, the number of AB complexes is arbitrarily set to 0 in the initial state (all `A`s are disconnected from `B`s in the initial mixture). In order to avoid this, one can modify the kappa file the following way: we set the initial concentration of `C` to 0 by deleting line 22. Now we introduce `C`s after 25 t.u by perturbation:

```
%mod: [T]>25 do $ADD 10000 C
```

The modified kappa file is available in the source repository, in the `model/` directory (file `abc-pert.ka`). Running again a simulation (a bit longer) by entering in the command line:

```
$ KaSim -i ABC-pert.ka -e 100000 -t 300 -o abc2.out
```

one obtains the curve given in Fig. 6.1

6.1.3 Snapshots

In the previous example, we let the system evolve for some time without its main reactant `C` in order to let other reactants go to a less arbitrary initial state. One may object that this way of proceeding is CPU-time consuming if one has to do this at each simulation. An alternative is to use the `$SNAPSHOT` primitive that allows a user to export a snapshot of the mixture at a given time point as a new (piece of) kappa file. For instance, the declaration:

```
%mod: [E-]/[E]>0.9 do $SNAPSHOT "prefix"
```

will ask KaSim to export the mixture the first time the ration of null events over productive ones will reach 0.9. The exported file will be named `prefix_`n`.ka` where n is the event number at which the snapshot was taken. Note that one may omit to define a prefix and simply type:

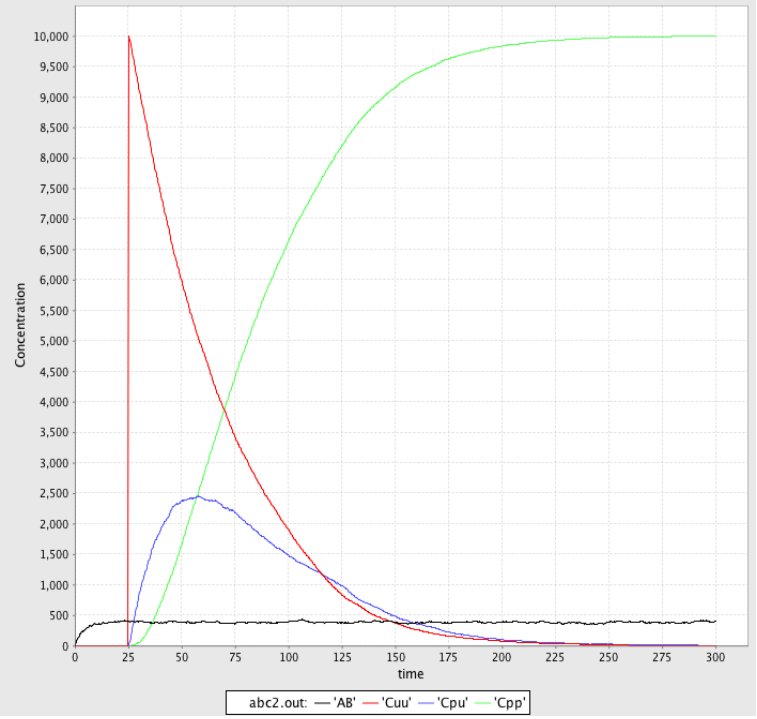

Figure 6.1: Simulation of the ABC model with a perturbation: for t<25s, only `‘a.b’` and `‘a..b’` rules may apply. This enables the concentration of `‘AB’` complexes to go to steady state, before introducing fresh `C`s at t=25s.

```
%mod: [E-]/[E]>0.9 do $SNAPSHOT
```

in which case the default prefix `snap` will be used for naming snapshots.

Note that if the name already exists a counter will be appended at the end of the file to prevent overwriting. Note that snapshots can be performed multiple times, for instance every 1000 events, using the declaration:

```
%mod: ([E+] [mod] 1000)=0 do $SNAPSHOT "abc" until [false]
```

which results in KaSim producing a snapshot every 1000 (productive) events until the simulation ends. The perturbation `$STOP "final_state"` will terminate the simulation whenever its precondition is satisfied and produce a snapshot of the last mixture. Note that instead of producing kappa files, one may use snapshot perturbations to produce an image of the mixture in the dot format using the parameter `"dotSnapshots"` (see Section 6.7).

6.1.4 Causality analysis

Causality analysis are always triggered by a perturbation of the form:

`%mod:` *start_ condition* `do $TRACK` *observable* `until` *end_ condition*

where *start_ condition* and *end_ condition* specify respectively when causality analysis should start and finish (remember that in the absence of `until` construct, causality analysis will only be performed during one computation event!). The *observable* is either a rule's name `'r'` or a variable name `'v'` pointing to a kappa expression. For instance, in the ABC model, the declaration:

```
%mod: [E=0] do $TRACK 'Cpp' until [false]
```

is asking KaSim to track the appearance of the observable `'Cpp'` since the beginning of the simulation, and display the causal explanation of every new occurrence of `'Cpp'`, until the end of the simulation. The explanation, that we call *causal flow*, is a set of rule application ordered by causality and displayed as a graph using dot format. In this graph, an edge `r`$\longrightarrow$ `r'` between two rule applications `r` and `r'` indicates that the first rule application has used, in the simulation, some sites that were modified by the application of the former. We show Fig. 6.2 an example of such causal flow.

Note that causal flow can show or hide more causal dependencies using various compression techniques that can be enabled as a simulation parameter (see Section 6.7).

6.2 Link type

In standard kappa, in order to require a site to be bound for an interaction to occur, one may use the *semi-link* construct `!_` which does not specify who the partner of the bond is. For instance in the variable:

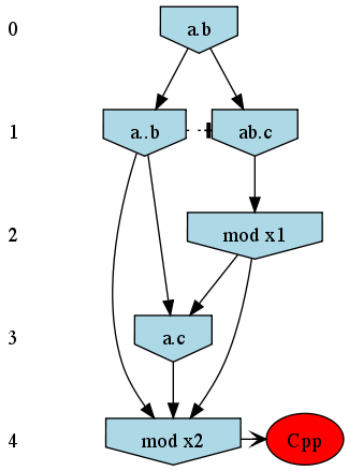

Figure 6.2: Causal flow for the observable `'Cpp'` of the ABC model. Plain arrows represent causal dependency, dotted arrows show asymmetric conflict between rule occurrences. Here the `'ab.c'` rule has to occur before the `'a.b'` rule. Red observable indicate that the last rule allowed one to observe a new instance of `'Cpp'`.

```
%var: 'ab' A(x!_),B(y!_)
```

will count the number of `A`s and `B`s connected to someone, including the limit case `A(x!1),B(y!1)`. It is sometimes convenient to specify the *type* of the semi-link, in order to restrict the choice of the binding partner. For instance the variable:

```
%var: 'ab' A(x!y.B),B(y!x.A)
```

will count the number of `A`s whose site `x` is connected to a site `y` of `B`, plus the number of `B`s whose site `y` is connected to a site `x` of `A`. Note that this still includes the case `A(x!1),B(y!1)`.

6.3 Implicit signature

KaSim permits users in a hurry to avoid writing agent signatures explicitly using the option `--implicit-signature` of the command line. The signature is then deduced using information gathered in the KF. Note that it is not recommended to use the DCDW convention for introduced agents in conjunction with the `--implicit-signature` option unless the default state of all sites is mentioned in the `%init` declarations or in the rules that create agents.

6.4 Influence map

The influence map of a model is an object that may help modelers checking the consistency of the rule set they use. It is generated by the command:

```
KaSim -i abc.ka -im map.dot -e 0
```

This will generate the so called *influence map* of the `abc.ka` file (the option `-e 0` specifies that no simulation should be run). The influence map is statically computed and does not depend on kinetic rates nor initial conditions. It describes how rules *with no side effect* may potentially influence each other during a simulation. KaSim will produce a dot format file containing the influence relation over all rules and observables of the model. The produced graph visualized using a circular rendering[1] is given in Figure 6.3. Observables are represented as circular nodes[2] and rules as rectangular nodes. Edges are decorated with the list of embeddings (separated by a semi-colon) allowing rules's right hand sides to be mapped to left hand sides. Note that the influence map will not display relations between rules that are induced by side effects (see Section 4.3.3).

More formally, consider the rules $r : L \to R$ and $s : L' \to R'$. One wishes to know whether it is possible that the application of rule r over a graph G creates a new instance of rule s. To do so, KaSim will try to generate a κ-term T containing a match for R and L' that overlap on some sites that are modified by r (see Figure 6.4 for illustration).

6.5 Flux map

The *flux map* is a powerful observation that tracks on the fly the influence rule applications have on each other. Contrary to the influence map, it is dynamically generated and tracks effective impacts (positive or negative) a every rule application. The flux map is obtained by the command:

```
KaSim -i abc.ka -im map.dot -t n
```

The resulting *flux map* is a graph where a positive edge between rules r and s (in green) indicates an overall positive contribution of r over s. Said otherwise, the sum of r applications increased the activity of s. Conversely, a negative edge (in red) will indicate that r had an overall negative impact on the activity of s. Note that visualization software permitting the importance of the flux between two rules can be observed by looking at the thickness of the edge between the rules. An example of flux map is given in Fig. 6.5. The generated dot file contains information about the importance of the negative or positive influence of a rule over another. This contribution is represented by the thickness of the edge that is proportional to the flux of activity that passed from one rule to the other.

[1] One may use for instance the `circo` program that is part of the *graphviz* suite.

[2] Observable can be viewed as identity rules for computing influence.

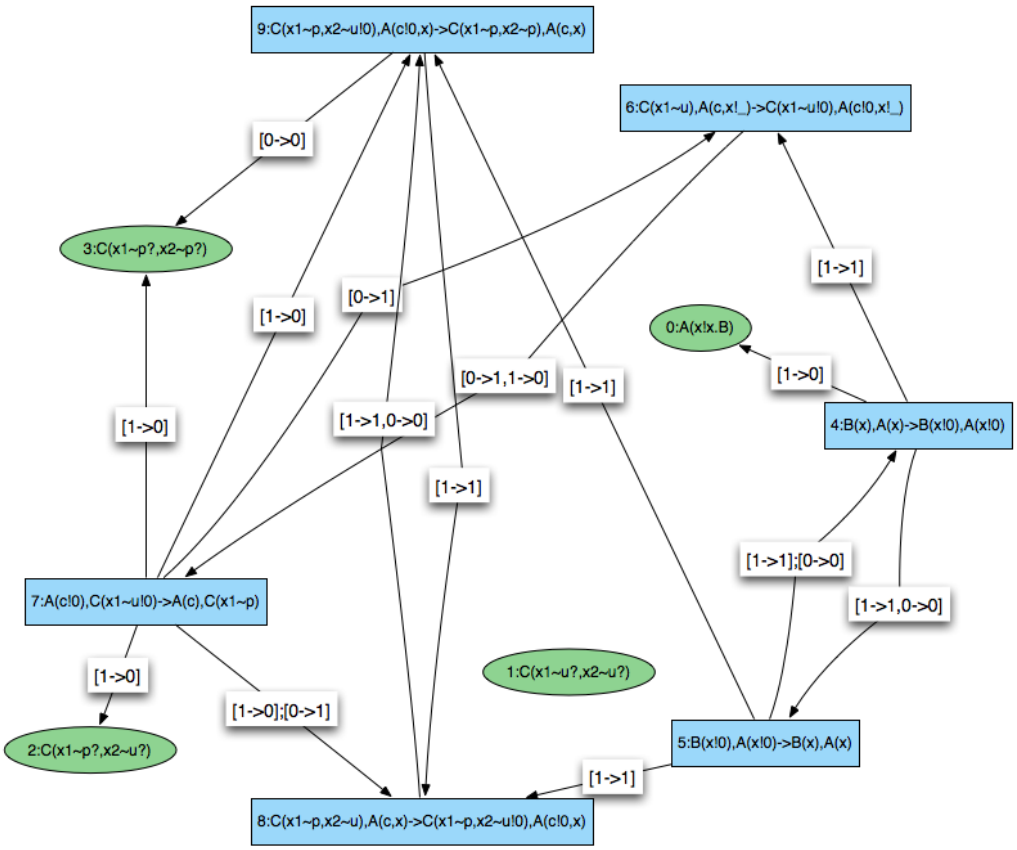

Figure 6.3: The influence map of the `abc.ka` model defined in Chapter 5. Note that observable `C(x1~u?,x2~u?)` is not activated by any other rule. This implies that its number of occurrences does not change during a simulation. Edge labels denote embeddings with the convention that $[i \to j]$ denotes the embedding of agent number i of the origin's rhs, into agent j of the target's lhs.

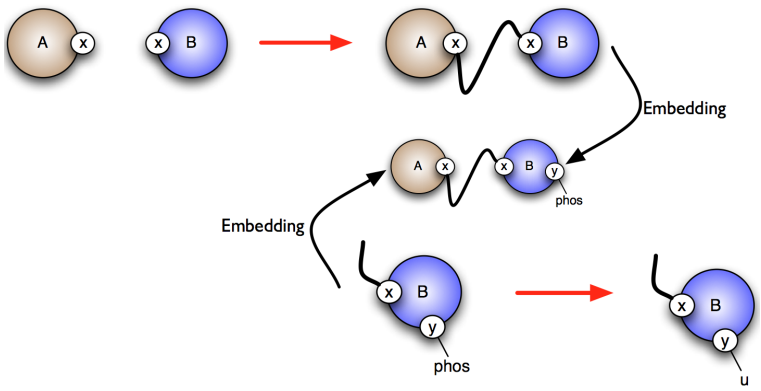

Figure 6.4: Computation of the influence of the top rule on the rule below: the right hand side of the first rules embeds in a common term with the left hand side of the second rule. It results that the first rule has a positive influence on the second.

Whenever the flux is too thin to be represented by a plain edge, it is shown dotted. Note that flux may vary during time, therefore the time or event limit of the simulation is of importance and will likely change the aspect of the produced map.

6.6 Simulation packages

The simulation algorithm that is implemented in KaSim requires an initialization phase whose complexity is proportional to $R*G$ where R is the cardinal of the rule set and G the size of the initial mixture. Thus for large systems, initialization may take a while. Whenever a user wishes to run several simulations of the *same* kappa model, it is possible to skip this initialization phase by creating a *simulation package*. For instance:

```
KaSim -i abc.ka -t n -make-sim abc.kasim
```

will generate a standard simulation of the `abc.ka` model, but in addition, will create the simulation package `abc.kasim` (.kasim extension is not mandatory). This package is a binary file, ie not human readable, that can be used as input of a new simulation using the command:

```
KaSim -load-sim abc.kasim -t k
```

Note that this simulation is now run for k time units instead of n. Importantly, simulation packages can only be given as input to the *same* KaSim that produced it. As a consequence, recompiling the code, or obtaining different binaries, will cause the simulation package to become useless.

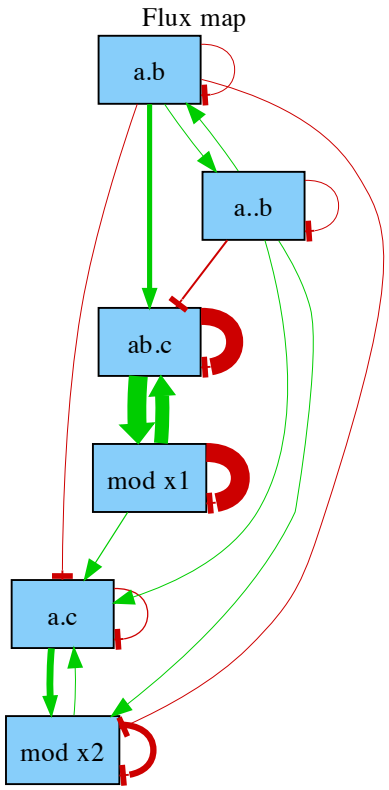

Figure 6.5: Flux map of the `abc.ka` model, taken from t=0 to t=20 time units. Note that the rules `ab.c` and `mod x1` are the biggest contributors to activity update in the simulated interval.

6.7 Simulation parameters configuration

In the KF (usually in a dedicated file) one may use expressions of the form:

`%set: "`*parameter_name*`" := "`*parameter_value*`"`

where tunable parameters are described table 6.2.

parameter	possible values	description
Causality analysis		
`"compressionMode"`	`"none", "strong", "weak"`	type of compression
`"cflowFileName"`	`"cflow.dot"` or any string	file name for causal flows
Pretty printing		
`"plotSepChar"`	`" "` or any character	separation character for plots
`"dotSnapshots"`	`"false", "true"`	generate dot snapshots
`"colorDot"`	`"false", "true"`	use colors in dot format files
`"progressBarSymbol"`	`"#"` or any character	symbol for the progress bar
`"progressBarSize"`	`"60"` or any integer	length of the progress bar
Simulation options		
`"dumpIfDeadlocked"`	`"false","true`	Snapshot when simulation is stalled
`"maxConsecutiveClash"`	`"2"` or any integer	number of consecutive clashes before giving up
		square approximation

Table 6.2: User defined parameters. Default values are given first in the possible values column.

Bibliography

[1] Vincent Danos, Jérome Féret, Walter Fontana, and Jean Krivine. Scalable simulation of cellular signaling networks. In *Proc. APLAS'07*, volume 4807 of *LNCS*, pages 139–157, 2007.

[2] Vincent Danos, Jérôme Feret, Walter Fontanta, Russ Harmer, and Jean Krivine. Rule based modeling of biological signaling. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *Proceedings of CONCUR 2007*, volume 4703 of *LNCS*, pages 17–41. Springer, 2007.

[3] Vincent Danos and Cosimo Laneve. Formal molecular biology. *Theoretical Computer Science*, 325, 2004.

[4] James R. Faeder, Mickael L. Blinov, and William S. Hlavacek. Rule based modeling of biochemical networks. *Complexity*, pages 22–41, 2005.

[5] Daniel T. Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of Computational Physics*, 22(4):403–434, 1976.

[6] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

Index